AFRL-RI-RS-TR-2010-29
Final Technical Report
January 2010

HIGH SPEED PUBLICATION SUBSCRIPTION BROKERING THROUGH HIGHLY PARALLEL PROCESSING ON FIELD PROGRAMMABLE GATE ARRAY (FPGA)

University of Missouri

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2010-29 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/
GEORGE RAMSEYER
Work Unit Manager

/s/
EDWARD J. JONES, Deputy Chief
Advanced Computing Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JANUARY 2010	Final	May 2007 – August 2009

4. TITLE AND SUBTITLE

HIGH SPEED PUBLICATION SUBSCRIPTION BROKERING THROUGH HIGHLY PARALLEL PROCESSING ON FIELD PROGRAMMABLE GATE ARRAY (FPGA)

5a. CONTRACT NUMBER
N/A

5b. GRANT NUMBER
FA8750-07-2-0166

5c. PROGRAM ELEMENT NUMBER
62702F

6. AUTHOR(S)

Chun-Shin Lin

5d. PROJECT NUMBER
AH09

5e. TASK NUMBER
UM

5f. WORK UNIT NUMBER
IZ

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Missouri
316 University Hall
Columbia, MO 65211-3020

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RITB
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
N/A

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2010-29

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 88ABW-2010-0158 Date Cleared: 15-Jan-2010

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Publish and subscribe approaches to information management systems reduce the complexity of these systems by loosely coupling publishers of information to subscribers of information through an intervening broker. Publications are typically written in Extensible Markup Language (XML) metadata, which a broker compares to XML Path Language (XPATH) predicates representing he subscriptions. Brokering matches result in the dissemination of a particular publication. Both the parsing of the American Standard Code for information interchange (ASCII) XML metadata and the evaluation of the logical predicates are time-consuming for programmable microprocessors. Augmenting programmable microprocessors with hardware acceleration from a Field Programmable Gate Array (FPGA) was determined here to reduce the latency and improved the throughput of this brokering. A hardware system was designed, built, and tested based upon utilizing (FPGA's) in the brokering process. Subscriptions were converted to hash values, and Look-Up-Table's (LUT's) were then synthesized on the FPGA's that corresponded to the hard value subscriptions. Ways to increase the reusability of LUT's were developed and demonstrated, which resulted in faster syntheses times, and an increase in the number of subscriptions possible on one FPGA. A software tool was developed and tested to completely facilitate automatic design generation for the FPGA syntheses. A method was developed and implemented to transfer multiple XML documents in group, which reduced the time delay in data transfer, the bottleneck in brokering, between the microprocessor and the FPGA. The brokering time of new publications was reduced to an average of 13μ sec.

15. SUBJECT TERMS
Information Management, Publish-Subscribe, Field Programmable Gate Array, Brokering, FPGA Syntheses

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			George O. Ramseyer
U	U	U	UU	50	19b. TELEPHONE NUMBER (*Include area code*) N/A

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 EXECUTIVE SUMMARY

Efficient and high speed Publication-Subscription brokering techniques, based upon the increased processing speeds afforded by reconfigurable Field Programmable Gate Arrays (FPGA's) tailored to specific applications, were investigated, and the results are presented here. Publish and subscribe approaches to information management simplified the construction of systems by loosely coupling publishers to subscribers through an intervening broker. Typically, the publications include Extensible Markup Language (XML) metadata that the broker compared to XML Path Language (XPATH) predicates representing the subscriptions to determine where to disseminate the publication.

Both the parsing of the American Standard Code for Information Interchange (ASCII) XML metadata and the evaluation of the logical predicates are time-consuming for programmable microprocessors. However, augmenting programmable microprocessors with hardware acceleration from an FPGA reduced the latency and improved the throughput.

An introduction to pub-sub brokering is presented, and then the basic ideas in the design are explained, along with additional details on how this was accomplished. Accomplishments from this effort included a new design, and the development and implementation of a microprogramming technique that identified XPATH's used in XML documents. The design was changed to increase the FPGA processing speed, and the clock rate was raised from 50 MHz to 133 MHz. A method was developed and implemented to transfer multiple XML documents in a group, which reduced the time delay in data transfer between the microprocessor and the FPGA. A software tool was developed and tested to completely facilitate automatic design generation. The documentation for the hardware and software of the developed design is included in the appendix to this report.

2 INTRODUCTION

The application of Field Programmable Gate Array's (FPGA's) for accelerating Publication-Subscription (Pub-Sub) brokering for information management has been investigated. Pub-Sub brokering (1) is a task of finding matches between subscriptions and a published item, and is the basis for some types of information management services. A published item could be an image from a satellite, information from sensors, a report regarding a specific area, etc. A publication includes a payload (such as an image) plus an XML descriptive document as a header. Since only the XML descriptive document is used in Pub-Sub brokering in this report, the word "publication" is used here to mean the XML metadata document. The XML document provides descriptive information about the published item, including, for example, the time the image was taken, the publisher's identification (ID), the payload format, and the geological location for the image.

Subscribers describe their needs using XPATH predicates in logic expressions. Pub-Sub brokering determines if the description in a publisher's XML satisfies the conditions in the subscribers' predicates. This task involves first parsing the XML document to extract information, and then evaluating all logical predicates. The task is typically done by software in information management systems such as the Joint Battlespace Infosphere (JBI) (2, 3) developed at the Air Force Research Laboratory (AFRL). This project investigates techniques of using hardware, and in particular FPGA's on the Heterogeneous High Performance Computer at AFRL, to accelerate Pub-Sub brokering.

Major efforts in this project have been to maximize the processing speed, the efficiency of hardware usage, and the system performance of a FPGA based computer for the brokering of publications and subscriptions. A method to reduce the required hardware area on a FPGA is presented, which led to a design capable of handling additional predicates on one FPGA.

A state machine to automatically generate predicates was developed that is capable of identifying XPATH's in an XML document. This ability addressed the following two constraints that the first design was required to comply with: (1) all possible leaves (metadata) for a publication type must be included in the XML document, and (2) those leaves in the XML document must appear in a specific order.

Techniques that increased the processing speed on an FPGA are then discussed. The clock rate was raised from 50 MHz to 133 MHz, and pipelining is an example of a technique that was also adopted. The Peripheral Component Interconnect (PCI) bus used a 133 MHz clock, and that clock was used in this new design. Experiments showed that using the same clock increased the reliability and stability of the FPGA. The method of transferring multiple XML documents in a group was then developed. The setup time for a Direct Memory Access **(**DMA**)** transfer was largely independent of the size of the transferred block, which resulted in reduced overhead timing when multiple documents were transferred together. A framework for a multi-node experimental system is also presented. This part of study: (1) defined the interface of the FPGA-based Pub-Sub brokering node with external nodes; (2) made the design more easily linkable to an upper level processor; and (3) evaluated the potential detail task management (adding/deleting predicates, etc.) with a Very High Speed Integrated Circuits Hardware Description Language (VHDL) re-synthesis, resulting in a FPGA reconfiguration for our design.

In Section 3 Pub-Sub brokering is introduced, which is the basis for the design developed here. In Section 4 the results of maximizing the efficiency of hardware usage and the processing speed are presented. A carefully constructed software tool that automatically generated the x86 file for configuring the FPGA is then reported. Suggestions for future work are presented in Section 5, and conclusions are presented in Section 6.

3 BACKGROUND AND THE BASIC APPROACH

This section introduces the Pub-Sub brokering problem under investigation and provides the basic ideas for the accelerator design on the FPGA. In Subsection 3.1 an example of an XML document together with coupled XPATH predicates describes the investigated problem. The overall design of the FPGA acceleration method is briefly introduced in Subsection 3.2, which explains how an XML document is parsed and how the parsed results are used. The method has the XML document downloaded to FPGA for parsing. The logic for parallel evaluation of predicates is configured into the FPGA for a very high-speed response. Details of subfunctions that were used in the design, such as the conversion of a variable-length character string into a fixed length hash value and the conversion of a number or a time into an integer, are presented in Subsection 3.3.

3.1 XML for Document Description and XPATH Predicates for Subscription Conditions

The Extensible Markup Language (XML) is a simple, very flexible text format originally designed to meet the challenges of large-scale electronic publishing. XML is also increasingly important in the exchange of a wide variety of data on the Web and elsewhere. World Wide Web Consortium (W3C) XML Schemas (4) provide mechanisms to define and describe the structure, content, and to some extent, the semantics of XML documents.

Figure 1 presents an example of an XML document (a) and four predicates (b) for a Pub-Sub application. Note that the XML document can be viewed as a tree structure. Each leaf is specified by an XPATH, e. g., "/metadata/IntelReportObject/Latitude." The option "Choice" available in XML schemas allows the generation of a subtree in different forms. One example is time, which can be either a time instant or a time period. The former is a subtree with a single node, and the latter is one with two nodes. In practical situations the number of leaves (terminal nodes) is likely to be tens or at most hundreds.

```
<metadata>
      <baseObject>
            <InfoObjectType>
                  <Name>mil.af.rl.mti.report</Name>
                  <MajorVersion>1</MajorVersion>
                  <MinorVersion>0</MinorVersion>
            </InfoObjectType>
            <PayloadFormat>text/plain</PayloadFormat>
            <TemporalExtent>
                  <Instantaneous>2003-08-10T14:20:00</Instantaneous>
            </TemporalExtent>
            <PublicationTime/>
            <InfoObjectID/>
            <PublisherID/>
            <PlatformID/>
      </baseObject>
      <IntelReportObject>
            <OriginatorID>VMAQ1</OriginatorID>
            <DetectionDateTime>20030728T163105Z</DetectionDateTime>
            <Latitude>42.538888888888884</Latitude>
            <Longitude>19.0</Longitude>
            <MTIObject>
                  <TrackID>000001</TrackID>
            </MTIObject>
      </IntelReportObject>
</metadata>
```

(a) An Example of an XML Document

```
(((/metadata/IntelReportOect/Latitude
>60)or(/metadata/IntelReportObject/Longitude <60))
and(/metadata/IntelReportObject/OriginatorID ='bravo'))

((/metadata/IntelReportObject/MTIObject/TrackID >17)
and(/metadata/IntelReportObject/OriginatorID !='alpha')
and(/metadata/IntelReportObject/Latitude
>45)and(/metadata/IntelReportObject/Longitude >45))

(((/metadata/IntelReportObject/Latitude
<45)and(/metadata/IntelReportObject/Longitude >=45)
and(/metadata/IntelReportObject/OriginatorID
!='delta'))or((/metadata/IntelReportObject/Latitude >=30)
and(/metadata/IntelReportObject/Longitude
<=90)and(/metadata/IntelReportObject/OriginatorID ='alpha')))

(((/metadata/IntelReportObject/Latitude
>45)or(/metadata/IntelReportObject/Longitude <60))
and(/metadata/IntelReportObject/OriginatorID ='VMAQ1'))
```

(b) Examples of Predicates

Figure 1. XML Document and XPATH Predicates

3.2 The FPGA Based Approach

The Field Programmable Gate Array approach is presented as the basis for matching subscriptions with publications.

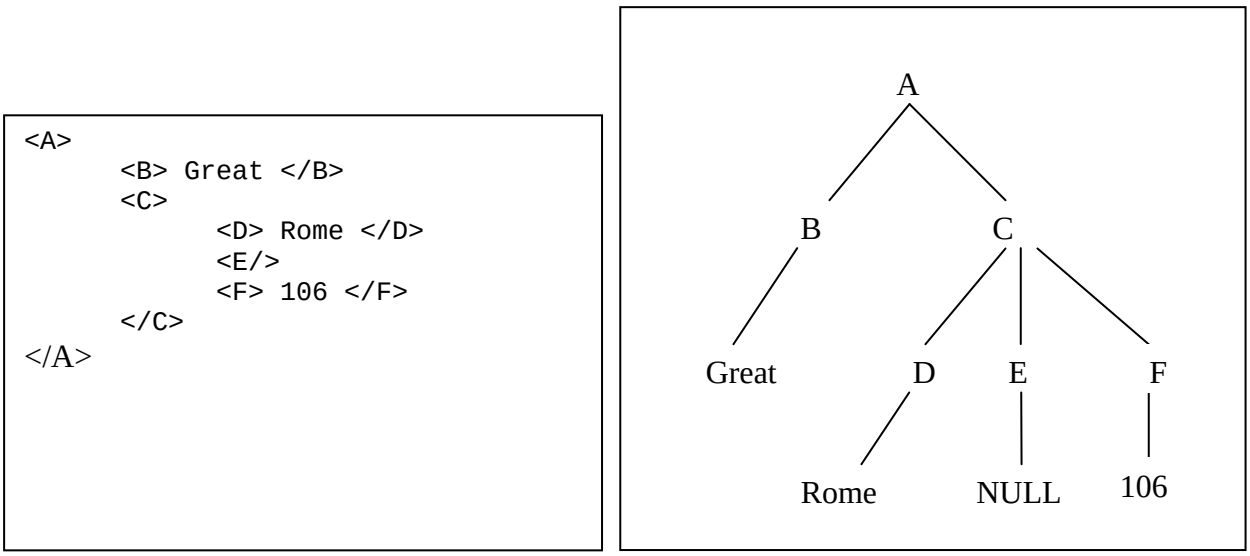
3.2.1 Overall structure and operations

When each FPGA acceleration module is dedicated to a specific type of publication, all the other XML documents associated with this specific type of publication must be generated using the same XML schema. For a newly published item, the XML document is downloaded from a microprocessor and stored into the FIFO (First In, First Out) on the FPGA board. The downloading is through a DMA transfer. A state machine on the FPGA walks through the document character by character to find the starting position of each leaf. For each leaf the offset of the starting point from the beginning of the document and the length of the string are recorded. This information provides rapid retrieval of the character string when there is a match. In addition to the pointer and string length, the FPGA also generates the converted data, which are hash values or converted numbers.

In this study, three types of commonly used data are considered. They are character strings, integers or numbers with a decimal point, and date/time. For a string of characters, the hash value offers a way for quick comparison to eliminate strings that do not match. A complete check using software is needed only when the hash values are equal. Integers, decimal-point numbers and date/time in an XML are represented in ASCII code, and converters convert them into binary integers for necessary inequality comparison required by predicates. Details on these conversions will be given in Subsection 3.3.

3.2.2 On board evaluations of logical predicates

In addition to sending the pointers, string lengths and hash values back to the microprocessor for necessary evaluations of predicates on the microprocessor, this design also evaluates predicates on an FPGA. A C++ program generates the necessary *Very High Speed Integrated Circuits Description Language* (VHDL) code from the primitive XPATH predicates. The predicates may be divided into subsets, each implemented in one submodule on the FPGA. Figure 2 shows an XML document and its tree representation. “Great” and “Rome” are both strings, while 106 is a number.



(a) An XML Document **(b) The Tree Representation**

Figure 2. Tree Representation of a Simple XML Document

There are four leaves specified by /A/B, /A/C/D, /A/C/E and /A/C/F, and in this example the leaves are assigned leaf numbers 0, 1, 2 and 3, respectively.

Three specific predicates using the Tree Representation in Figure 2b are illustrated in Figure 3.

/A/B = “Greg” AND /A/C/D != “Syracuse”
/A/C/D = “Utica” OR /A/B = “Great”
/A/C/F < 106

Figure 3. Illustrative Predicates

After the the leaves are assigned leaf numbers, the above predicates in Figure 3 can be translated into a representation that includes the leaf number, logic/arithmetic relationship, the hash value or converted number, and a logic operator. This new representation is presented in Table 1.

Table 1. Hash Values of Predicates

Leaf #	Relationship	Hash/binary number	Logic Operator
0	=	Hash value of “Greg”	AND
1	!=	Hash value of “Syracuse”	End_Predicate
1	=	Hash value of “Utica”	OR
0	=	Hash value of “Great”	End_Predicate
3	<	Binary representation of 106	End_Predicate

The FPGA parser is a state machine which parses the XML, sending the hash values and lengths, or the binary numbers, into registers. That information is then broadcast to buses. Predicate evaluation logic will access these data for the necessary evaluations. For the previous example with four leaves, the parser will store the following information into four registers, as shown in Table 2.

Table 2. Parser Register Values

Register 0	16-bit hash value of “Great”	5	(16-bit string length for “Great”)
Register 1	16-bit hash value of “Rome”	4	(16-bit string length for “Rome”)
Register 2	0	0	(16-bit string length for NULL string)
Register 3	Converted datum for 106	(32 bits)	

The predicate VHDL generator converts the primitive XPATH predicates into VHDL code, which is then integrated with the parser to form a reconfigurable processing unit for very high speed predicate evaluation. A segment of VHDL code is shown in Figure 4.

```
boolean(0) <= ‘1’ when leaf_reg(0) = "01010111101110010000000000000100" else ‘0’;
boolean(1) <= ‘1’ when leaf_reg(1) = "10001100100000010000000000001000" else ’0’;
predicate(0) <= boolean(0) AND boolean(1);
boolean(2) <= ‘1’ when leaf_reg(1) = "11101010010111001000000000000101" else ‘0’;
boolean(3) <= ‘1’ when leaf_reg(0) = "10010100100100010000000000000101" else ’0’;
predicate(1) <= boolean(2) OR boolean(3);
predicate(2) <= ‘1’ when leaf_reg(3) = "00000000000000011001111000010000" else ‘0’;
```

Figure 4. VHDL Code for Clause Evaluations

The Boolean values in Figure 4 are the result of the clause evaluations, and the leaf_reg's are the registers storing the information shown in Table 2 (converted number or hashed value with string length). The evaluation results are returned to the microprocessor. With the results, the evaluations of a large number of subscriptions by software are eliminated. It is expected that only a very small subset of subscriptions will need to be thoroughly checked. A high-speed evaluation response is expected from the high speed parsing of XML, the highly parallel predicate evaluations on FPGA's, and from efficient filtering.

To determine if a character string in a clause matches the one in the XML document, the logic only compares the hash value and the length. If they don't match, then the two strings are positively different. If they match, the two strings are just "possibly" the same. In this implemented design, a 2-bit result is obtained for each clause; one bit is the Boolean result and the other indicates if the result is "sure." The same 2-bit representation is used for inequality checks.

3.3 Major Design Sub-functions Required

The major design sub-functions are presented.

3.3.1 Hash function for converting character string

A parser implemented on an FPGA and the predicate VHDL generator generates hash values in the same way. The hash value is initially set to 0. When a new ASCII code (a character in the leaf portion) is received, 4-bit blocks of the binary number are swapped in a way as shown in Figure 5, and then the new 7-bit binary value (ASCII code) is added to it. The hash function is selected because of its simple hardware implementation.

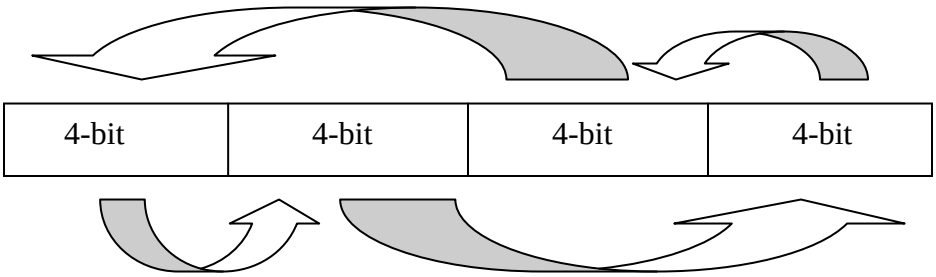


Figure 5. Swapping in Hash Value Computation

3.3.2 Converting ASCII numbers to binary representation

This converter is for numerical data within the range $[-2^{31}/1000, (2^{31}-1)/1000]$. The data is multiplied by 1000, and only the integer part is kept. For truncated numbers, when a comparison shows that one number is greater than another, the result is surely correct, but there is ambiguity when the comparison shows two truncated numbers are equal.

The 7-bit ASCII codes for numbers 0 to 9 are 30, 31, … 39. With the first three bits dropped, we have the Binary-Coded Decimal (BCD) code of that digit. The binary number generator takes the following steps to generate the binary number:

- Initialize the binary number to 0.
- For each retrieved digit, add it to 10 times the current binary number representation.
- The process continues until the whole string is completed or three digits beyond a decimal point.
- The number is then timed by 1 (no decimal point observed), 10 (one digit after the decimal point seen), 100 or 1000 to get the right result.

For example, if the number is 19.4, the state machine will read in 1, multiply it by 10 and add 9, skip the decimal point, multiply 19 by 10 and add 4, and then multiply 194 by 100 at the end. If the number is 42.77291, the result will be 42772.

Multiplication of a number Y by 10 is done by calculating 8*Y + 2*Y. Since the multiplication of 2 is a single bit shift and the multiplication of 8 is a shift of 3 bits, the multiplication is performed by one adder. Multiplication of 10 twice gets a multiplication of 100. However for a multiplication of 1000, 1024*Y – 16*Y – 8*Y is used, which requires two additions (subtractions). The propagation delay for such an implementation is shorter than that by multiplying 10 three times; signal propagation is through two adders instead of three.

3.3.3 Time stamp conversion

One possible format for a time stamp is YYYYMMDDHHmmSS±HHmm, where Y, M, D, H, m, S represent digits for the field for year, month, date, hour, minute, and second, respectively. The part ±HHmm is for time adjustment, and is usually required for handling a time zone difference. In the conversion a reference year, for example Year 1980, is first determined. With the reference year used, the year is represented in the range [-32, 31], which can be specified in 6 bits. The month, date, hour, minute and second can be specified in 4, 5, 5, 6 and 6 bits, respectively. The total number of bits is 32. In parsing, a small state machine is required at the end to do the time adjustment (±HHmm). Note that the update may be propagated backward from minute to hour, date, month and year. A 0 to 5 additional clock cycles may be required to finish the time adjustment.

3.4 Hardware System

The Testbed that was utilized for this study was the Heterogeneous High Performance Computer (HHPC) with Annapolis Micro Systems' WILDSTAR™II FPGA boards that was located at the AFRL Information Directorate. Each of the 48 nodes has a dual 2.2 GHz Virtex™-II processing element plus 12 M FPGA's, 4 Gbytes of SDRAM, Mvrinet 320 MB/sec interconnects, and 80 GB disk space, all on an Adaptive Computing Board.

4 TECHNIQUE OPTIMIZATION AND EXPERIMENTAL RESULTS

The optimized technique and experimental results are presented here.

4.1 FPGA Hardware Synthesis

The efficiency of the hardware synthesis was increased, and is described here.

4.1.1 Synthesis Design Considerations

When the number of predicates in a set is increased, additional comparators are required, which results in an increase in the number of modules for predicate evaluations. This results in an increase in the required FPGA's hardware resources (area utilization or hardware usage) required for the analysis. A careful arrangement in the design can significantly reduce the hardware usage. To maximize the efficiency of the FPGA resource usage, a method was developed that allowed for the efficient sharing of comparators for predicates evaluation.

Initially the predicate strings to be evaluated were compared with the 32-bit values each as one datum. Area minimization was performed by the Xilinx software, but this minimization optimization became less efficient as the logic complexity grew. To maximize the sharing, each 32-bit value was divided into eight 4-bit pieces. Each 4-bit piece required at least one 4-input look-up table (LUT4) for comparisons, and the comparators could then be reused as often as necessary. Some types of data types required two LUT4's. The reusable LUT4 were set up for every possible comparison that can occur. This meant that 128 comparisons (16 possible binary values for each of the eight 4-bit pieces) were set up for each of the different leaves in the XML document. The Xilinx software automatically connected only the comparisons that were actually used in the predicates evaluation, and discarded the rest.

For numerical comparisons, an approach similar to that of the strings was taken. Every possible 4-bit numerical comparison was set up, and then the Xilinx synthesizer disconnected those that were not used. Since every numerical comparison could result in one of three outcomes (equal, less than, greater than), a two bit value was passed to the predicate module to ensure that the correct result was known. This meant that the numerical register was twice the size of the string register due to the two-bit logic. Every possible four-bit numerical comparison was created in a separate VHDL file (Comparisons.vhd in Figure 6), and then the predicate module(s) was (were) connected to only the lookup tables that were required.

The comparisons were done using an overloaded '<' operator that returned values according to the following scheme: equals - '10', less than - '01', greater than - '11'. These two bits were then processed by the predicate module(s) using a specially written function (to be shown later) that started at the most significant bit and moved downward to determine which type of two bit logic the function should return based on whether the comparison was false-sure (01), true-sure (11), or true-unsure (10). Numerical '=' comparisons were handled the same way as strings; a Boolean value was returned for each group of four bits, and then all of them were ANDed together. Figure 6 gives the sample code for shared comparisons and the overloaded less-than operator.

```
num(0) <= breg(3 downto 0) < "0000";
num(1) <= breg(3 downto 0) < "0001";
num(2) <= breg(3 downto 0) < "0010";
.
.
.
num(127) <= breg(31 downto 28) < "1111";
```

(a) Sample Code for Numerical Comparisons (128 comparators; num(): 2-bit)

```
equals(0) <= breg(3 downto 0) = "0000";
equals(1) <= breg(3 downto 0) = "0001";
equals(2) <= breg(3 downto 0) = "0010";
.
.
.
equals(63) <= breg(15 downto 12) = "1111";
```

(b) Sample Code for String Comparisons Using 16-bit Hash Value (64 comparators)

```
function "<" (X,Y: std_logic_vector(3 downto 0))
return std_logic_vector is variable Z : std_logic_vector(1 downto 0);
begin

if(X > Y) then Z:= "11";
elsif(X = Y) then Z:= "10";
else Z:= "01"; end if;

return (Z);
end "<";
```

(c) Overloaded Less-than Operator

Figure 6. Shared Comparators and the Overloaded Operator

In the developed system a large set of predicates was divided into subsets, and each subset was then evaluated in a sub-module. This significantly reduced the complexity in the hardware synthesis, and thus reduced the synthesis time. To maximize the benefits of the shared approach, the reusable 4-bit comparisons were separated from the predicate sub-modules, and became individual modules. This allowed comparisons to be created once and then used by all of the other predicate sub-modules. A port was attached to transfer the data between the modules, and Xilinx software then discarded the unused connections and/or tables. When the predicates were divided into ten sub-modules, more of the FPGA area was consumed than for a single module, but the synthesis time was much shorter, and the total hardware usage was reduced (compared to the case not using shared comparators).

A block diagram of the hardware connections for this design is presented in Figure 7. The parser (parser.vhd) sent out the hashed/converted leaf values to shared comparators (comparisons.vhd; all 4-bit comparators) through "bus register". The comparison results were then sent to the predicate evaluators (two in this example). The evaluation results were sent back to the paser and transferred to the microprocessor. This setup can be expanded to support as many predicate submodules as desired, and Xilinx software will disconnect any unused bits on the port connections.

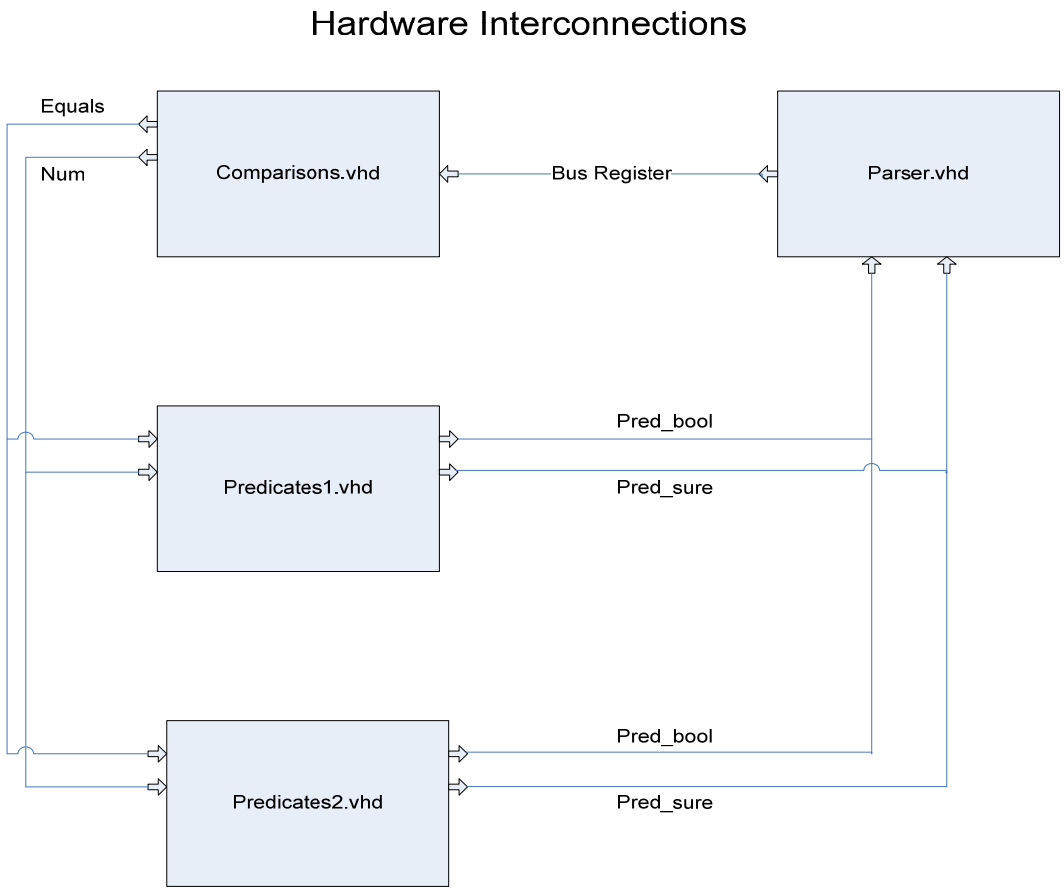


Figure 7. Block Diagram of Hardware Interconnections

Each type of comparison requires a certain number of LUTs inside the predicate module (predicates#.vhd in Figure 7), and a certain number inside the comparisons module (conversions.vhd in Figure 7). Tables inside the comparisons module can be reused, thereby saving space. Inside the predicate module, Xilinx software performs some optimizations as well by packing unfilled tables together.

Table 3. shows the theoretical LUT usage for basic comparisons, clause evaluations and predicate evaluations in modules comparisons.vhd and predicates#.vhd. The last four entries are based on how many clauses are involved in the evaluation of each predicate.

Table 3. Theoretical LUT Usage

Case	LUT inside Predicate	LUT inside Comparisons
String =	4 per clause	8 (for 32 bits)
Numerical =	4 per clause	8
Numerical >	10 per clause	16
Numerical <	10 per clause	16
<=4 clauses/Pred	2	0
<=7 clauses /Pred	4	0
<=10 clauses /Pred	6	0
>10 clauses /Pred	8+	0

The original greater-than function listed in Figure 8 operated on the entire 32-bit number at once and used the greater-than operator from the *std_logic* package to determine which type of Boolean-sure logic to return. This was an efficient approach on a per comparison basis, but for a large number of comparisons there was an opportunity to reuse many look up tables already created.

By implementing this reusable approach for both types of comparisons, FPGA hardware usage can be reduced by a considerable amount. Configurations that previously did not fit on the chip can now be implemented. The new design approach allowed many more predicates to be included on a single FPGA.

```
function ">" (X,Y: std_logic_vector(31 downto 0))
   return std_logic_vector is variable Z : std_logic_vector(1 downto 0);
begin
                    if(X > Y) then Z:= "11";
                    elsif(X = Y) then Z:= "10";
                    else Z:= "01"; end if;

                    return (Z);
                    end ">";
```

Figure 8. Original Greater-than Comparison Function

Figure 9 lists the greater-than function that evaluates the less-than, greater-than, or equal data for each set of 4-bits from the comparisons.vhd module and determines if the entire number comparison is greater. The function then returns a two-bit logic value based on whether the clause is true-sure, true-unsure, or false-sure. As indicated in Table 3, 10 LUTs are needed for the implementation of this function.

```
function greater (A,B,C,D,E,F,G,H: std_logic_vector(1 downto 0))
   return std_logic_vector is variable Z : std_logic_vector(1 downto 0);
begin
if A = "11" then Z := "11";
elsif A ="10" then
        if B = "11" then Z := "11";
        elsif B = "10" then
                if C = "11" then Z := "11";
                elsif C = "10" then
                        if D = "11" then Z := "11";
                        elsif D = "10" then
                                if E = "11" then Z := "11";
                                elsif E = "10" then
                                        if F = "11" then Z := "11";
                                        elsif F = "10" then
                                                if G = "11" then Z := "11";
                                                elsif G = "10" then
                                                        if H = "11" then Z := "11";
                                                        elsif H = "10" then Z := "10";
                                                        else Z := "01";
                                                        end if;
                                                else Z := "01";
                                                end if;
                                        else Z := "01";
                                        end if;
                                else Z := "01";
                                end if;
                        else Z := "01";
                        end if;
                else Z := "01";
                end if;
        else Z := "01";
        end if;
else Z := "01";
end if;
```

Figure 9. Greater-than Comparison Function

4.1.2 Synthesis Implementation

The synthesized hardware was implemented on the FPGA's, and when run, reads XML documents, does the necessary hashing, and evaluates the predicates. The first step in a synthesis was to design the configuration of the FPGA's for implementation, and then hardware was assembled for that configuration. The results of several hardware syntheses are presented in Table 4. Table 4(a) describes hardware syntheses when 4-LUT comparators were not optimized utilizing techniques described in Section 4.1.1, and Table 4(b) describes syntheses when the 4-LUT comparators were optimized. The Hardware syntheses without the 4-LUT Comparators (Table 4(a)) used 32-bit comparisons. All syntheses were run at 133 MHz.

The Character Strings were the number of different unique strings ("Rome", "Smith", ...etc.) in that synthesis' subscriptions. The specific clauses in the predicates were determined randomly, and had a 50% probability of being either a numerical comparison or a string. For the Synthesis Case A, for example, there were 586 clauses. Thus, nominally 293 clauses were numerical comparisons, and 293 were strings. The 500 predicate scenarios C and F were numerical comparisons only.

The predicate subscriptions were then randomly constructed clauses, which were formed from the possible strings and numerical comparisons. The number of clauses was nominally 6 per predicate, although as few as 2 clauses, and as many as 8 clauses, were constructed in predicates.

Table 4 (a). **Hardware Syntheses (baselines) without 4-LUT Comparators Optimization**

Synthesis Case	Subscriptions				Results		
	Predicates	Sets	Unique Character Strings	Clauses	Comparators	Area (%)	Time (s)
A	100	1	50	586	7367	10.9	406
B	100	1	300	566	7922	11.7	393
C	500	1	0	6629	*	*	*
D	100	10	50	586	8331	12.3	393
E	100	10	300	566	8252	12.2	380
F	500	10	0	6629	47550	70.3	4273
G	1000	10	100	9398	72256	106.1	*
H	1000	10	400	9297	75498	111.7	*
I	2000	10	500	13126	105257	155.7	*

(* indicates no data available).

(b). Hardware Synthesis for Optimized 4-LUT Comparators

Synthesis Case	Subscriptions				Results		
	Predicates	Sets	Unique Character Strings	Clauses	Comparators	Area (%)	Time (s)
A	100	1	50	586	6112	9.0	355
B	100	1	300	566	6294	9.3	373
C	500	1	0	6629	30997	45.8	21863
D	100	10	50	586	6646	9.8	440
E	100	10	300	566	6705	9.9	400
F	500	10	0	6629	32904	48.7	3328
G	1000	10	100	9398	36789	54.4	4139
H	1000	10	400	9297	37908	56.1	4457
I	2000	10	500	13126	48367	71.6	*

(* indicates no data available).

The times in Tables 4(a) and 4(b) are for the time it took for the complete hardware synthesis of each case, and included the design of the configuration, the implementation of that design, and the implementation of the parser on the FPGA. The Comparators are the number of 4-LUT that were designed and implemented for each synthesis case, and the Area is the percentage area for the FPGA's that were used in the design. In cases where the design area was more than 100%, that design could not be implemented. In every case where there was actual data taken, the Optimized Comparators took less area and were synthesized in less time than for the baseline comparators.

The usefulness of utilizing sets is clearly demonstrated in comparing the results of Synthesis Cases C and F in Table 4(b) for the Optimized Comparators. The total synthesis time was 21863 seconds (6 hours) for one set of predicates, but when the predicates were divided into 10 sets, the time of synthesis was slightly less than one hour. The area used on the FPGA was 2.9% more for the 10 set case than the single case. When there were only 100 predicates, as in Cases B and E in Table 4(b) for the Optimized Comparators, it actually took more time to synthesis the 10 sets than the single set.

In Figure 10 is presented a comparison of the FPGA Area required for the subscriptions divided into 10 sets for the baseline (red) hardware syntheses (Table 4(a)) and the optimized (blue) hardware syntheses (Table 4(b)). As the number of predicates increased, the number of LUT4's required for the optimized predicates was reduced by more than 50%.

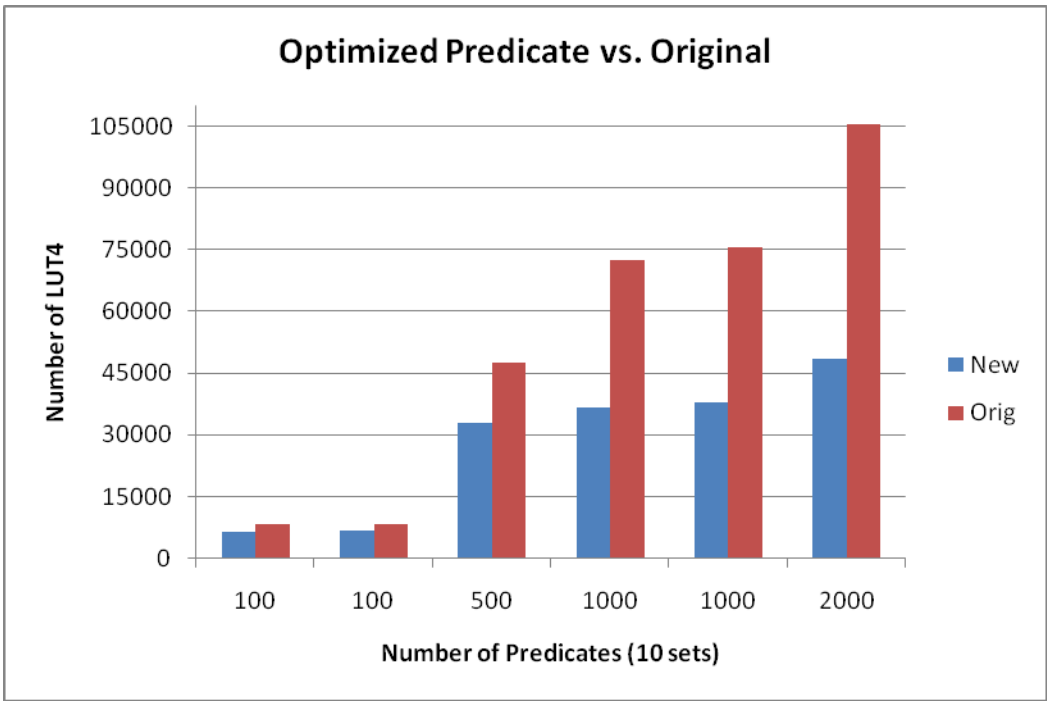


Figure 10. FPGA Area Usage Comparison

4.2 State Machine for Leaf Identification

The original description of the design of a perfect hash was to map each XPATH into an assigned leaf number. The drawback in this method was that a perfect hash map is difficult and time consuming to manually develop when the number of leaves goes beyond 50 or so. A mapping microprogram based state machine was developed to replace the previous perfect hashing design. A state machine is a model of the behavior of a finite number of states, transitions between those states, and actions. It is similar to a "flow graph" where the way in which the logic runs can be inspected when certain conditions are met. A computer program toolbox has been developed to automatically take the list of leaves and create the microprogram for this application. The procedure is done automatically, and the whole design is user friendly.

A simple example for explaining this mapping microprogram creation is presented. The simple example has two leaves, as shown in Figure 11.

```
/AB/C,1
/AB/D,2
```

Figure 11. Two Leaves in a Microprogram

A C++ program generated the tree presented in Figure 12 for this set of leaves.

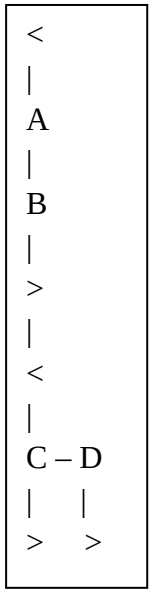


Figure 12. Set of Leaves in a Tree.

| symbolizes a bottom pointer, - symbolizes a same level pointer

Each node in the tree consists of: (1) Name: The character that represents this node; (2) String (leaf) #: (If this is the last node of the leaf name string, this will be the number related to this string. Otherwise, it will be 0); (3) Node #: Each node will have a number, which will correlate to its memory address in the microprogram; (4) Same Level Pointer: A pointer to another node that has the same previous character, but different current character; and (5) Bottom Pointer: A pointer to the next character in this leaf, or NULL if it is the last character in a leaf. The contents of each node for this example are presented in Figure 13.

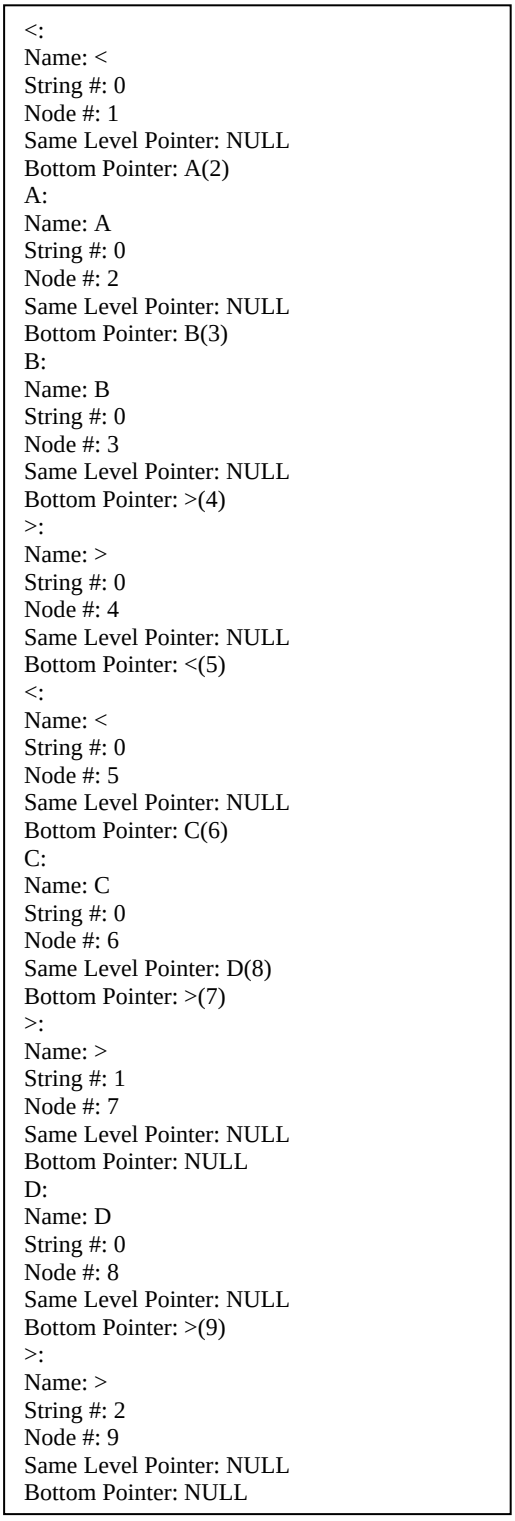


Figure 13. The Contents of Each Node

The mapping microprogram has a control word consisting of: Character : PS : False Address : String Number : String Found. When executed, Character is compared with the input and the result determines whether the control will increment the address or jump to the False Address. A non-negative string number is provided if the state machine detects a leaf name from the input string based on the String Found bit. PS is a bit used to push the current address into a stack for later usage. The microprogram presented in Figure 14 was constructed from the tree data example that began with Figure 11.

```
< : 0 : 11 : -1 : 0
A : 1 : 11 : -1 : 0
B : 0 : 11 : -1 : 0
> : 0 : 11 : -1 : 0
< : 0 : 11 : -1 : 0
C : 1 : 8 : -1 : 0
> : 0 : 11 : 1 : 1
D : 1 : 11 : -1 : 0
> : 0 : 11 : 2 : 1
0 : 0 : 10 : 0 : 0 – leaf found
0 : 0 : 11 : 0 : 0 – error
```

Figure 14. Constructed Mapping Microprogram.

Note that two extra lines are added at the end of the constructed mapping microprogram. The first extra line is for the situation when a string is found. The second is for when an error has occurred. Given the schema for a type of publication, this software tool can automatically generate the microprogram-based state machine and integrate it with other parts of designs for Pub-Sub brokering.

4.3 Increased Processing Speeds

The first generation design has been carefully examined and bottlenecks causing longer processing delays have been identified. That design was modified to reduce time delays to achieve a 133 MHz clock rate. The original clock rate was 50 MHz. The following subsections report the major changes of the design.

4.3.1 Reducing Time Delay from FIFO to Parser

To reduce the time delay from the input to the parser, a cache mini-buffer was placed between the state machine and the parser. The mini-buffer (MINI FIFO) had only two registers, and received data from the input FIFO and then transferred that data to the parser. Figure 15 shows the structure with a MINI FIFO added. Using this MINI FIFO, the delay was reduced from 5ns to 2ns.

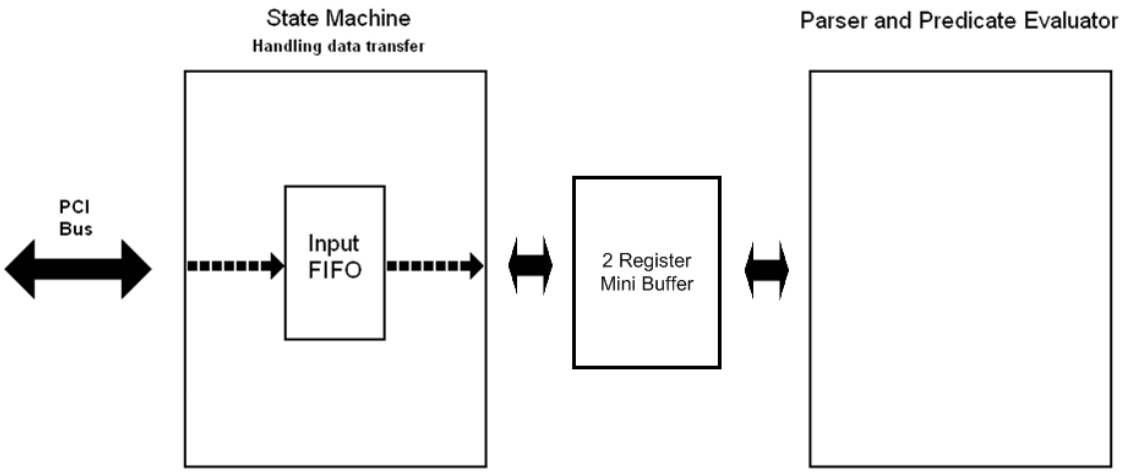


Figure 15. Cache Mini- Buffer (MINI FIFO)

4.3.2 Conversion of a number in ASCII into a 32-bit binary number

In the conversion of an ASCII number to a 32-bit number, that ASCII number to be converted must be multiplied by 1, 10, 100, 1000, -1, -10, -100 or -1000. This was originally implemented using two adders/subtracters for each case, and with a multiplexer to select one out of the eight results (see Figure 16). Originally, 1024N, 16N, etc. were generated by simple data shifting implemented by proper signal wiring. The delay of this processing through two adders/subtracters and one multiplexer was 18 ns. The new design has one set of registers inserted after each layer of adders/subtracters. This increased this rarely used computation by two clock cycles, but greatly reduced the clock period for all operations.

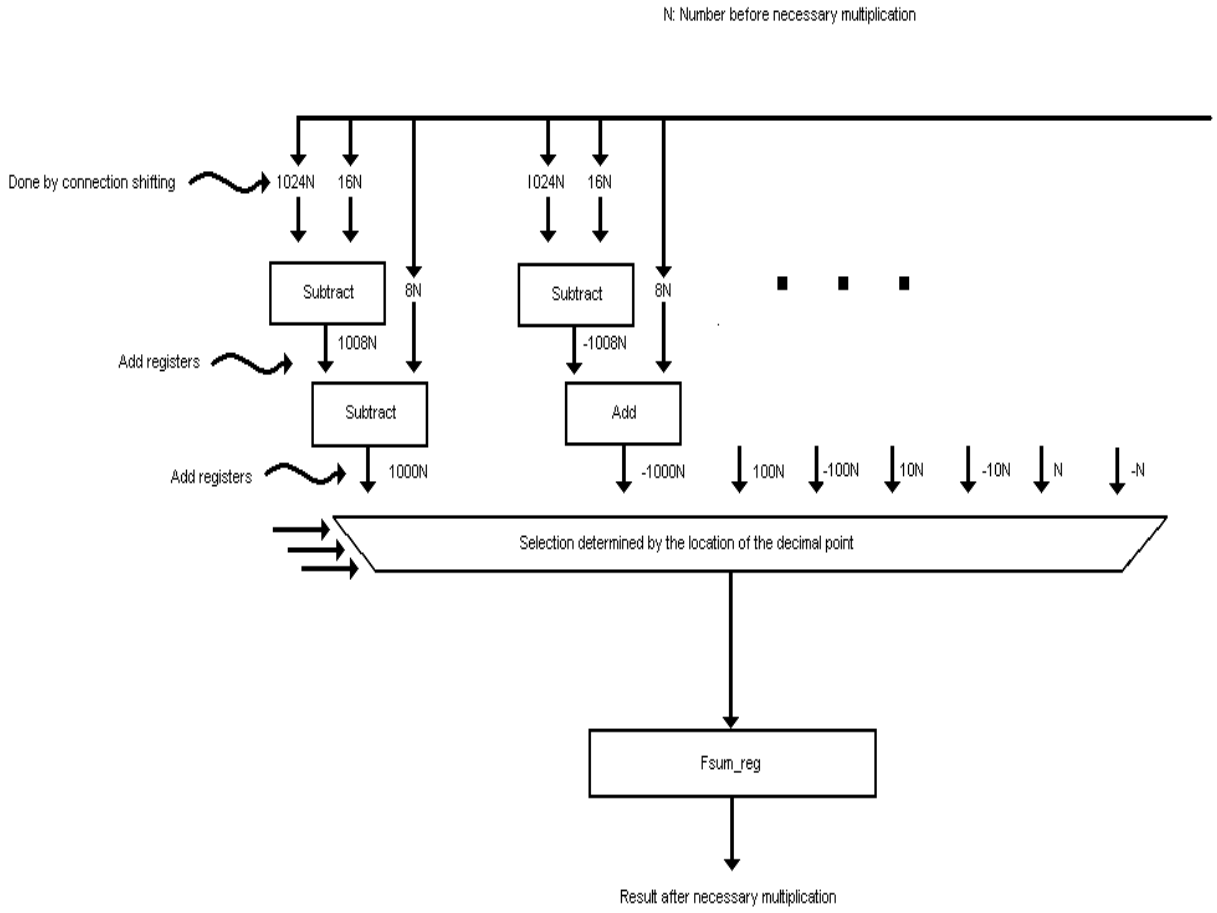


Figure 16. Number Conversion

4.3.3 Delay in date/time correction

The date/time is dependent upon time zones. In the worst possible case, a change of one hour can require changes in the day, the month and even the year. For instance, it could be on New Year's Eve in one time zone the morning of the New Year's Day in another time zone. To check for this condition previously required a complicated condition check, which caused long delays. To reduce this delay, the complicated condition check was divided into two clock cycles, resulting in a shorter clock period.

4.3.4 Delay in accessing microprogramming control words

The microprogramming setup uses Block Random Access Memory (BRAM) to store a tree of data for comparison to the data coming in from the Peripheral Component Interface (PCI) bus. Decisions are then made based on this comparison as to which piece of data to load from the tree next and also what the next state should be in the XML document parser. For maximum efficiency and speed, it is necessary to parse one incoming character per clock cycle. For a

character to be processed, it is necessary to wait until the data is ready on the data bus, compare this data to the input character, and then put the next address onto the address bus of the BRAM. It was found that the time required to complete all of these steps was roughly 10ns, resulting in a maximum speed of about 100 MHz. A large portion of this delay was due to the long "clock to output" time of the BRAM (approximately 2.65ns), which corresponded to the amount of time it took for data to become available on the data bus after the clock edge had occurred. There were also long routing delays on both the address and data nets of the BRAM, most likely due to congestion. These routing delays were usually on the order of 1-2 ns for both the address and data connections. To eliminate these delays, registers were placed on the address and data lines of the BRAM to isolate it from other logic. This effectively reduced the path from the data output of the BRAM to the comparison logic, and from logic back to the address bus, in half.

This was accomplished by implementing a pipeline structure in which the address was placed into the address register in the first cycle, the data was available on the output bus the following cycle, and then in the final cycle the data was stored in a register which was then ready for comparison. All of these steps occur in a parallel pipeline structure so that one character was still processed per clock cycle. For this to work, a two cycle delay allowed the pipeline to fill, and was required upon startup and whenever there was a mismatch between incoming data and the tree data (i.e., a stack jump needed to occur). The pipeline operated under the expectation that the next address to be loaded from the BRAM was one greater than the current address, which was the case the majority of the time. Exceptions were when there was stack activity. When it was necessary to pull from the stack and go back to a previous address, the number of wasted cycles depended on how far back the jump went and also on the structure of the XML document. Two cycles were required for every time the program needed to jump to get to the correct position in the microprogram. These concepts are illustrated in Table 5.

Table 5. **Block Random Access Memory (BRAM) Pipelined Data Access**

Clock Cycle	Address Register	Data Out	Data Register
0	000	-	-
1	001	A	-
2	002	B	A
3	003	C	B
4	004	D	C
5	005	E	D

After this pipeline structure was implemented, it was determined that the design speed was still limited by another path involving block RAM resources. The path from the input FIFO through logic to the mini-FIFO in the parser was slightly under 9 ns. This long path was also due to the "clock to output" time and net routing delays of the BRAM that the input FIFO was implemented in. The path between these two FIFOs passed through combinational logic that converted the 64-bit FIFO data into 8-bit chunks of ASCII data for the parser to use. By registering the data at this point, it was possible to split this path, which allowed the overall speed to be increased. The registering of the data requires an initial two cycle delay while the first datum from the input FIFO was read in and stored, and was then capable of providing one 8-bit data to the mini-FIFO every clock cycle thereafter. This conversion process served as an interface between the input and parser FIFO's, and broke the timing path between them.

4.4 Transfer of XML Documents in Groups

One drawback in the FPGA coprocessor approach was the time needed for communications between the host and the FPGA. For each transaction there was some amount of overhead time required for sending data to and receiving data from the FPGA through the Direct Memory Access (DMA). Reducing this amount of wasted time was desirable to maximize the efficiency and speed of the coprocessor approach. It was determined that when a DMA transaction was initiated, data was sent to the FPGA in 256-byte bursts, each of which was separated by 110-115 cycles of inactivity. It has also been determined that the data transfer was 32 bits at a time instead of the 64 bits that was expected. Both of these details had minimal effect on the overall transaction time since the FPGA state machine began as soon as the first 64 bits of data arrived.

Since the state machine required one clock cycle to process each character (8-bit), the state machine was not able to catch up to the DMA process and caused a buffer under-run. Thus the main portion of the overhead time was the time required *to initiate the DMA transfer* and *to handle the response from the FPGA*. It was noted that sending either a single XML document or a group of them required the same amount of time for transfer initiation. Thus, by sending multiple XML documents to the FPGA and receiving the results all at once, the average processing time per publication could be reduced. The only factor limiting how many XML documents can be sent at once was the size of the input FIFO. Currently, both the input and output FIFOs each can hold a little over 8,000 characters. A larger capacity input FIFO will allow the average DMA overhead to be decreased further.

An additional reduction in the DMA overhead time resulted from the optimization of the host program. Originally, a string of several function calls was executed to perform the DMA transaction and return the response. By removing the excess functions and directly calling the Wildstar API function that performs the transaction, it was possible to reduce this overall time.

Table 6 illustrates the benefits of sending as many publications as the FIFO can hold. This test was conducted at 133 MHz with XML documents with an average size of 1,150 characters each. An XML document of this size would take approximately 10 μs to process on the FPGA. It can be seen that the more XML documents that are sent at a time, the closer the per-publication time approaches the actual processing time (10 μs) on the FPGA, eliminating redundant DMA overhead.

Table 6. **Processing Times for Multiple XML Documents Simultaneously Processed**

XML Documents Processed	Total Processing Time*	Per-Publication Time
1	18.5 μs	18.5 μs
2	29.4 μs	14.7 μs
3	40.7 μs	13.4 μs
4	53.5 μs	13.3 μs
5	65 μs	13.0 μs
6	77 μs	12.8 μs

*Time includes all DMA transfers

4.5 Framework for a Multi-node FPGA System

The Pub-Sub brokering design presented in the previous sections was executed on a single-node on the Heterogeneous High Performance Computer (HHPC). It allowed a specified number of publications to be sent to the FPGA to be checked. Any unsure predicates were then checked via the microprocessor after the FPGA had reported the results. This design could be a submodule in a complete information management system, but an interface to the upper level module would be necessary. In this part of study, this is explored.

The presumed multiple-node Pub-Sub system requires a central node to manage the distribution of publications to multiple client nodes, which would each parse the data and return matches. This would allow the system to operate with more predicates than the FPGA on a single node could process. Predicate management functions would allow the addition and deletion of predicates while the system is running. The duties of re-synthesizing the FPGA image when enough subscription changes occurred is incorporated by using another node that was specifically dedicated to this purpose.

A minimum of three nodes are then required to run this system. The central node must be started first, followed by the rebuild node, and then the client node. Once all three nodes have been started, commands can be given to the central node to have it carry out various pub-sub functions. The functions of the three nodes are summarized:

1) Central node (C node)
   a) Send commands to the Pub-Sub processing node
      (1) Predicates
         (a) Add a new predicate
         (b) Delete an existing predicate
      (2) Process a new publication
   b) Receive results from the Pub-Sub processing node
   c) Command the FPGA re-synthesis node to re-synthesize the design for the newest set of predicates.

2) FPGA re-synthesis client node (S node)
   a) Receive the re-synthesis command from the central node
      (1) Acquire the data/information for the new design
      (2) Generate the re-synthesized design (i.e., x86 file);
   b) Send the x86 file to the Pub-Sub processing client node
      (1) Reconfigure its FPGA board
      (2) Implement the updated design

3) Pub-Sub processing client node (P node)
   a) Receive commands from the central node
      (1) Add a new predicate
      (2) Delete a predicate to add or delete a predicate
   b) Publication
      (1) Process a publication
      (2) Send a request to the central node for returning the Pub-Sub brokering results
   c) Maintain necessary information for the valid predicates and their ID's
   d) Receive a new x86 file and reconfigure the FPGA
   e) Do other necessary initialization/updating.

Sockets were used for communications between the various nodes in the system. The connections were oriented in a star topology with the central node as the hub. Error checking was used to monitor the connections, and an error caused the nodes to shut themselves down if the socket was broken. Error checking also monitored transfers over the sockets to ensure that the entire message was successfully sent. Since the socket functions could return before all of the data had been sent, they were repeatedly called using the remaining data until the entire message had been transmitted. All of the socket calls used in this program were blocking, meaning that they would halt the system and then wait for the complimentary call to be issued on the other end of the socket before proceeding.

The central node operated through the use of commands that instructed the program as to which pub-sub action to execute. The commands were either entered one at a time through the console or listed sequentially in a file and passed to *stdin* at runtime. The commands allowed the user to add/delete predicates, parse publications, and manage re-synthesis tasks for all of the client nodes. Publications were parsed by giving the program the name of an XML file, which would then be opened, and all of the publications in the file were then checked against all of the predicates in the client node(s). Predicates were added by specifying the XPATH file name of the new predicates, as well as which client node the predicates should be added to. Predicate deletions were done by listing the ID numbers for all of the predicates that were to be removed. The re-synthesis command caused the system to begin to rebuild the x86 image of a particular node. At that time the rebuild process was simulated by renaming an existing x86 file that had been previously created. If there were changes to the predicates that had not been incorporated into a hardware rebuild before the system was shut down, that information was written to a file, and the changes were then implemented the next time the system started. This happened if, for example, predicates were added or deleted and then the system immediately shutdown.

To parse publications the central node read the specified number of publications from the XML source and sent them to all client nodes. The number of publications that were sent at a time was set by a command-line option. After sending the pubs to each client, the central node waited for a response from each node. Each client node packaged the incoming pubs and then sent them to the FPGA in one DMA transaction. After the results were returned by the FPGA, the client node packaged all of the results together and returned them to the central node. Note that the values returned as hits were no longer the index values of the place where the predicate resided in the predicate list, but were now a unique ID assigned to each predicate. These ID numbers were attached to each predicate in the XPATH file at the start of the line.

There have been some input file changes for the multi-node version of the system. Instead of ‘predinfo.pcf’, the new system needed the file ‘preds_client#.xpath’, where # is the ID number of the client node the file corresponded to. The x86 image also followed a similar naming convention, *i.e.* ‘pe_client#.x86’. The ‘leaves.txt’ file also needed to be uploaded so that the XML tree structure could be extracted. Please refer to the Appendix for detailed information.

When a rebuild process was started, all predicates flagged for deletion were simply removed, and all new additions were added onto the end of the list. This produced some issues concerning how fast the system could be rebuilt. By keeping all of the system changes in only one or two predicate modules, the re-synthesis was completed in a much faster time period. If a predicate at the beginning of the predicate XPATH file was deleted, it caused the modules boundaries to shift, and therefore all of the predicate modules had to be rebuilt. This was avoided by keeping all volatile predicates at the end of the XPATH file, and only placing predicates that were not expected to be deleted at the beginning.

This part of study achieved the following: (1) The interface of the FPGA-based Pub-Sub brokering node was defined with external nodes; (2) the design was improved for rapidly linking to an upper level processor; and (3) The framework was developed for testing the design including the detail task management (adding/deleting predicates, etc.), VHDL re-synthesis, and FPGA reconfiguration.

4.6 Software Tool for Automatic Generation of the Design

The procedure to construct the design is very complicated to perform manually. Without a proper tool to automatically generate the design, the programs developed in this work would be difficult to build. A script file with the use of makefiles was developed to help perform this complicated process. Given a set of predicates and the schema that was used to generate XML documents, the script file can be run to automatically generate the x86 file for configuring an FPGA. The documentation for the developed system, including the script file and how to use it, is included in the Appendix.

5 FUTURE WORK

This section lists some issues that have been identified for possible improvement.

1) Using NGCBuild during (re)synthesis in addition to NGDBuild could speed up resynthesis time for data sets with large numbers of leaf nodes. When the number of leaf nodes in a data set becomes large, the interfaces between predicate modules and the parser will also become very large (the size of the interfaces scales directly with the number of leaves). These large interface ports cause the NGDBuild step of the synthesis process (which incorporates all modules into a single netlist) to slow down considerably. If it is known that only one or two predicate modules are likely to have changes, it is possible to use NGCBuild to incorporate all of the other, non-changing modules into a single netlist, and then allow NGDBuild to simply incorporate the changed modules into the netlist. The NGC netlist is only created once, and then all subsequent resynthesis (that only has changes in the dynamic modules) can be done faster by allowing NGDBuild to not duplicate previous work, and simply add in the changed modules.

2) The WildStar-II in the HHPC has 2 Virtex-II FPGAs on each board, but the current software implementation only utilizes one of the FGPAs per node. Changes could be made to allow the publication to first be sent to one processing element and then the other, followed by the combining of the results. This could double the predicate capacity of each node of the HHPC. The biggest hurdle is how to handle which predicates are in which FPGA while the system is running, for resynthesis purposes, etc.

3) To help speed up resynthesis times, it would be useful to have a system in place for optimizing the predicates file before the rebuild process is started. When predicates are added to or deleted from the system, it is possible for the sub module groupings that were established during the initial synthesis to shift and cross boundaries. If this happens, the resynthesis process will be forced to re-synthesize the entire design instead of only a few changed predicate modules. Currently, newly added predicates are just put at the end of the list, and deleted ones are simply removed. Therefore, removing the first predicate in the system would cause the entire design to be resynthesized, because all groupings would shift upward by one predicate. An organization system that upheld sub module boundaries in the XPATH file could greatly reduce resynthesis times after add/delete operations.

4) The current software checking portion of the host program (that handles the checks for recently added predicates that haven't made it into the FPGA yet), processes data 32 predicates at a time. The way the software check is currently written requires that full software checks be used for the lowest multiple of 32 that is greater than the number of software predicates. This means that some predicate results that were returned by the FPGA must be software checked

again in order to be sure that all software-only predicates actually got checked. This process unnecessarily wastes time by duplicating work that has already been completed.

6 CONCLUSIONS

The study on improvement of hardware usage made it possible to reduce the hardware area by 50%, enabling the FPGA to accommodate more predicates. XPATH for a leaf can be identified while an XML document is being parsed. This removes some restrictions that the old design needed to comply with. The processing speed was increased in two ways. The hardware design was refined to increase the clock rate from 50MHz to 133MHz, which made it possible to use a single clock for PCI bus and this design. The single clock made the operation much more stable and reliable than having separate clocks. The speed was also increased through the transfer of multiple XML documents in a group. The setup time for a transfer was almost independent of the document size, greatly reducing the overhead time. To take care of the complexity in constructing the design, a software tool to automatically generate the design has been developed. A script file with the use of a couple makefiles has been provided to help perform the complicated process. The design from this project will be a submodule in a complete information management system. Thus an interface to the upper level module is necessary. We have explored a multiple-node Pub-Sub system that has a central node to manage the distribution of publications to multiple client nodes which would each parse the data and return matches. This study offers a suggestion for the interface of the FPGA-based Pub-Sub brokering node with external nodes. The complete software is provided to the Air Force Research Lab at Rome and the documentation for the design is provided in Appendix.

7 REFERENCES

1. K. Birman and T. Joseph, “Exploiting Virtual Synchrony in Distributed Systems”, Proc. 11th ACM Symp. Operating Systems Principles, pp. 123-138, Dec. 1987.

2. Scientific Advisory Board, “Building the Joint Battlespace Infosphere Volume 1: Summary,” SAB-TR-99-02, 2000.

3. Scientific Advisory Board, “Report on Building the Joint Battlespace Infosphere Volume 2: Interactive Information Technologies,” SAB-TR-99-02, 1999.

4. http://www.w3.org/.

8 ACRONYMS

Acronym	Definition
AFRL	Air Force Research Laboratory
API	Application Programming Interface
ASCII	Atlas Transformation Language
BRAM	Block Random Access Memory (in FPGA)
C++	A General Purpose Programming Language
DMA	Direct Memory Access
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
HHPC	Heterogeneous High Performance Computer
JBI	Joint Battlespace Infosphere
LUT	Look Up Table
MHz	Megahertz
ns	Nanosecond
PC	Personal Computer
PCI	Peripheral Component Interconnect
RAM	Random Access Memory
VHDL	Very High Speed Integrated Circuits Hardware Description Language
W3C	World Wide Web Consortium
XPATH	XML Path Language
XML	Extensible Markup Language

APPENDIX
PUB-SUB BROKERING DESIGN DOCUMENTATION

The PubSub brokering system is meant to allow the pairing of publishers and subscribers at high speed in an FPGA by checking predicate queries from the subscribers against the data made available by publishers. This documentation provides an overview of the functionality of the system and how to use it. Section 1 focuses on the hardware required for multiple nodes, Section 2 on the hardware for a single node, Section 3 on the software for multiple nodes, and Section 4 on the software for a single node

Section 1. Hardware Required for Multiple Nodes

The user inputs to the PubSub system are: an XML file containing the publication data; an XPATH file containing the subscriber predicates; and a file containing information on the paths and leaf nodes in the XML tree. For details on the format of these files, see the following subsection.

Input Format: Sample files are included in sample_files directory. There are several important things to note about formatting:

Publications:

- Need to be in XML format
- Lines should be terminated with Unix style new line characters
- Each individual pub is terminated by an EOT character
- There should be two EOT characters following the final pub
- Quotation marks are not used for strings/dates
- Due to the way the FPGA is set up, there needs to be ~15 characters between the last leaf data and the EOT. This is to allow the last data to settle before trying to read the result
- The only acceptable dateTime formats are as follows:
  YYYYMMDDTHHMMSSZ
  YYYYMMDDTHHMMSS+HHMM (or -HHMM)
  The T is optional, but the parser will not work with dashes or colons

Sample publication format:

```
<Tagged_Record_Extensions>
<Last_Calibration_Date>20041004T220534+0600</Last_Calibration_Date>
 <MTI_Report_Extension>
  <AC_Altitude>10000</AC_Altitude>
```

```
    <AC_Heading>28</AC_Heading>
    <Side_Of_Aircraft>L</Side_Of_Aircraft>
    <No_Valid_Targets>79</No_Valid_Targets>
    <Target>
     <Number>466</Number>
     <Location>
      <Latitude>28.3671847</Latitude>
      <Longitude>-096.0236283</Longitude>
     </Location>
     <Location_Accuracy>547.1</Location_Accuracy>
     <Radial_Velocity>-192</Radial_Velocity>
    </Target>
   </MTI_Report_Extension>
  </Tagged_Record_Extensions>
```

Predicates:

- Predicate lines start with an identifier number, followed by a comma, and then the predicate
- Strings and dateTimes need to be enclosed in quotations
- There are a wider variety of dateTime formats that are acceptable
- Pretty much anything except YYYY-M-D is ok

See date.c (in xpxlate folder) for more details

Sample predicates entries:

```
115,/Tagged_Record_Extensions/MTI_Report_Extension/AC_Heading<351
116,/Tagged_Record_Extensions/MTI_Report_Extension/Target/Number=292
117,/Tagged_Record_Extensions/MTI_Report_Extension/ No_Valid_Targets<10
118,/Tagged_Record_Extensions/MTI_Report_Extension/Side_Of_Aircraft="R"
119,/Tagged_Record_Extensions/MTI_Report_Extension/AC_Altitude<60000
```

Leaves.txt:

- Format is one per line of path/leaf,datatype
- Make sure to use the right data type labels here. Wrong data types will generate junk results.
    - 0 = string
    - 1 = number
    - 2 = DateTime

Sample leaves.txt entries

```
/Tagged_Record_Extensions/Last_Calibration_Date,2
```

/Tagged_Record_Extensions/MTI_Report_Extension/AC_Altitude,1
/Tagged_Record_Extensions/MTI_Report_Extension/AC_Heading,1
/Tagged_Record_Extensions/MTI_Report_Extension/Side_Of_Aircraft,0
/Tagged_Record_Extensions/MTI_Report_Extension/No_Valid_Targets,1
/Tagged_Record_Extensions/MTI_Report_Extension/Target/Number,1
/Tagged_Record_Extensions/MTI_Report_Extension/Target/Location/Latitude,1
/Tagged_Record_Extensions/MTI_Report_Extension/Target/Location/Longitude,1
/Tagged_Record_Extensions/MTI_Report_Extension/Target/Location_Accuracy,1
/Tagged_Record_Extensions/MTI_Report_Extension/Target/Radial_Velocity,1

A brief overview of the steps that occur during the synthesis process is presented.. There are three questions that the Synthesis script will ask when it is started; how many predicate subsets to use, whether or not to use the shared comparators (more in a later subsection), and what number should be used to identify the output files.

1) The input predicate set is generally split into smaller subsets. This allows for faster synthesis, as well as faster resynthesis later on. The downfall of having multiple subsets is that more space on the FPGA will be consumed because of the inability to share resources. It is generally a good idea to try to group predicates according to predictions for future changes.

2) After the predicates have been divided into subsets, they are converted from XPATH format to a VHDL file. They are then synthesized into modules to be used later.

3) Synthesis then occurs on the parser module, as well as other static system components. These modules generally only need synthesized once for any particular publication format, so during subsequent resynthesis processes, these steps will be skipped. The microprogramming module is also created here according to the format of the publications.

4) Synplify Pro is then run to create a top level interface file that will bond all of the lower level sub modules together. This file may or may not need to be created during each resynthesis, depending upon the changes made.

5) At this point, the entire design can be incorporated into a single netlist by bringing all of the modules together based on the interfaces specified in the file created by Synplify.

6) The netlist can now be mapped into technology components on the FPGA.

7) Place and route can now make the connections between the various logic components.

8) The final design is then converted into an x86 format, which can then be programmed into the FPGA.

There are several helper programs that are used to perform various functions during synthesis; documentation for each program can be found in the ./sorce directory. If any portion of the synthesis process needs to be changed, generally one of these programs is going to need to be modified.

When large numbers of predicates are used, timing can become an issue when the design is being routed. The ‘pe.ucf’ file contains constraints that lock down the location of the block RAM modules and set an area group for the top-level part of the design. The current setup yielded the best results seen, but it is possible to experiment with this file and tweak the design further. There is also a constraint in this file that allows the combinatorial path to/from the predicate modules to take longer than one clock cycle.

Incorrect input to the synthesis process can cause it to not complete, or to produce an FPGA image which will not give correct results. The leaf information file must contain the correct data types for each path. For example, setting the data type to integer on a path that really contains a string will result in erroneous data being returned by the FPGA. Typos involving formatting (such as missing quotations) will also most likely cause the synthesis process to fail. It is also very important that the publication file uses Unix style (\n) newline characters only. The parser does not support windows style (\n\r) newlines, and will stall repeatedly. If the synthesis process is interrupted by an error or the user before completing, it is usually a good idea to execute the 'clean' script (after fixing the problem, of course) before attempting to synthesize again. When the process is interrupted, it may not have successfully completed certain steps that will be omitted if the process is immediately started again.

The synthesis process can be run with or without the use of the shared comparators approach. If this option is turned on, the comparator logic blocks will be pulled outside of the predicate modules and placed in a separate module. This can help to save FPGA area by not duplicating logic, but it can also slow down the synthesis process. For data sets with large numbers of leaf nodes in the XML structure, it can take a very long time to synthesize the comparator module (it only needs to be done once for each XML structure, but is still slow). For example, synthesizing the comparator module alone on a data set with 60 leaf nodes takes over an hour.

The synthesis script is set up so that during resynthesis, any predicate module that hasn't changed is skipped. This also occurs for other steps in the script, so that only the affected predicate module is resynthesized. The entire design can then once again be incorporated into a single

netlist, mapped, placed, and routed. Xilinx also provides the ability for the place and route process to be incremental and only changed parts of the design will be rerouted. This can significantly reduce resynthesis times when only changing a few predicates. The incremental design approach works best when all changed predicates are in the same module, so care should be taken to try to ensure this is the case in the beginning.

A change was made to the predicate file format to allow the addition of a unique identifier number for each predicate. The number is placed before the start of each predicate line and is followed by a comma. By adding this to the predicates file, the values returned by the FPGA can be mapped to a unique identifier that allows the matching predicate to be more easily located.

After the synthesis process has been completed, pe.x86, pubs.xml, preds.xpath, and leaves.txt (see new naming conventions below) should be uploaded to the HHPC. The software program requires at least 3 nodes on the HHPC to run. One node acts as the main server which handles incoming publications and manages the workhorse nodes. Another node serves the purpose of providing an environment for resynthesis processes to occur. There can then be one or more client nodes which handle the actual parsing of publications, as well as managing a software database of predicates. Commands can be issued to the server for things such as parsing publications, adding predicates, deleting predicates, and performing resynthesis. The server will then handle these tasks by utilizing the other nodes as appropriate. Incoming publications are sent to all clients to be parsed, so that the system can handle more predicates than what will fit in a single FPGA. The server demo program accepts commands through *stdin*, or a list of commands in a file can be piped in at runtime. Explanations of the commands can be found in the server_test.c source code.

Note that the HHPC no longer requires 'predinfo.pcf' to be uploaded, and instead uses preds.xpath. The same info that was contained in 'predinfo.pcf' is extracted from the XPATH file at startup. Also note that the preds.xpath file on the HHPC always represents the exact state of the FPGA hardware at any point in time.

Since the HHPC operates in a shared storage environment, when the FPGA on more than one node is being used it is necessary to have different filenames for each node (i.e. there can't be two pe.x86 files with identical names). Therefore, a naming convention was established to identify which node the various input files belong to. This naming scheme only applies to the x86 and XPATH files, and is as follows: pe_client#.x86 and preds_client#.xpath where # is the node identifier number indexed from zero. So for example, the files for the third client node should be named pe_client2.x86 and preds_client2.xpath. The leaves.txt file and any publication XML files do not have any changes.

Care must be taken when choosing which nodes will be used to compile and run the program. The compile process will not work on all nodes due to library issues, so some trial and error may be required to find a working node for building the program. Likewise, the FPGA does not work on all nodes, so any node running the client portion of the software must be running on a node with a working FPGA.

The synthesis script is intended to work with Xilinx version 9. Slight changes may be required to make it work with a different version. The program gamkef is responsible for creating the secondary synthesis script, and therefore may need to be modified to get the process to work with a different version.

If the FPGA always stalls when sending it publications, make sure the publication is in the proper format and that Unix style newlines are being used.

Section 2. Hardware Required for a Single Node

All of the information in the multi-node hardware documentation applies to the single-node system as well, except for Predicate ID's, Files to Upload, and HHPC Execution. The single-node version of the system does not use XPATH files with predicate IDs at the start of each line. Instead of ID numbers being returned by the FPGA, an index value is returned that shows the position of the hit in the XPATH file. The files created by the synthesis process that need to be uploaded to the HHPC are pe.x86 and predinfo.pcf. The naming conventions for the multi-node version do not apply to the single-node case, and the input files should be left with these original names. Note that the single-node version requires predicates.pcf, not preds.xpath, to be uploaded. After uploading the correct files to the HHPC, the test program can be ran which will parse the specified XML publications file and return the results.

Section 3. Software for Multiple Nodes

The host software requires at least four input files to run:

1. pe.x86 - image to load into the FPGA
2. preds.xpath - an XPATH listing of all the predicates in the FPGA image
3. leaves.txt - a listing of all possible paths in the XML tree and the corresponding data types
4. pubs.xml - an XML document containing publications

There may be multiple x86 and XPATH files if there will be more than one client node processing publications. There can also be multiple XML publication files that will be parsed,

but there can only be one leaves.txt since the whole system must operate on only one type of publication structure at a time. Also, see the file naming conventions in the hardware section.

The test server application provides the following functionality:

1. Parse publication file - After specifying an input file name, publications are sent to all available client nodes, and then the returned results are printed out. The references that are returned correspond to the predicate ID numbers that are included at the start of each line in the predicate XPATH files.
2. Add predicates - After specifying an input file name and which client node to add them to, the predicates will be loaded into the software database on that particular node and will then be available for future pub checking and hardware rebuilds.
3. Delete predicates - The node to delete the predicates from is first specified, followed by a comma separated list of predicate IDs to remove. The deleted predicates will not be removed from hardware until the next rebuild on that node, but they will be ignored in all subsequent checks until that time.
4. Initiate rebuild - The server can initiate a rebuild on any client node which will cause all of that node's current, active predicates to be built into a new FPGA image.
5. Wait for rebuild - If necessary, the server can block and wait until the rebuild process is completed for any particular node.

Commands are given to the server program through *stdin*, or can be piped in at runtime. The command is followed by one space and then the arguments.

1. Parse publication = 1 <XML_file> (*i.e.* '1 pubs.xml')
2. Add predicates = 2 <node_number> <pred_file> (*i.e.* '2 0 new_preds.xpath')
3. Delete predicates = 3 <node_number> <pred_ids> (*i.e.* '3 0 104,118,190')
4. (Pred ids for deletion are separated by commas, with no spaces, and no comma at the end.)
5. Start rebuild = 4 <node_number> (*i.e.* '4 1')
6. Wait for rebuild = 5 <node_number> (*i.e.* '5 1')
7. Terminate program = 6

An example command file:

1 pubs1.xml	=>	Parse publications from file 'pubs1.xml'
2 1 new_preds1.xpath	=>	Add the predicates in 'new_preds1.xpath' to node 1
3 0 114,118,119	=>	Delete preds with ids 114,118,and 119 from node 0
1 pubs2.xml	=>	Parse publications from file 'pubs2.xml'
4 0	=>	Start a rebuild on node id 0
5 0	=>	Pause until the rebuild on node id 0 is finished
1 pubs3.xml	=>	Parse publications from file 'pubs3.xml'
6	=>	Terminate program

Not all nodes on the HHPC will successfully compile the host software due to library issues, so it may be necessary to try several nodes before finding one that will successfully build the executable. The FPGA does not always work on all nodes either, so if error messages are generated when trying to communicate with the FPGA, try a different node. At least three nodes are required to run the host demonstration program. One node runs the main server, one runs the rebuild process, and at least one runs the client software. The provided Makefile will generate all three executables. The command lines to launch each node are as follows:

./server_test [num_pubs_to_send] [number_of_clients] < [command_file]

./rebuild [ip_address_of_server]

./test [ip_address_of_server]

When the server is first launched, it waits for the rebuild node to connect first. The program must be started on the rebuild node before any clients; otherwise the program is not going to work. The server will then wait for the specified number of clients to connect. The num_pubs_to_send argument specifies how many publications will be sent in each transmission to the client nodes. The proper input files must be in the working directory for the program to run. Also make sure that the proper naming conventions have been used for input files.

Due to the current inability to run the synthesis script on the HHPC (license issues, etc...), the demonstration program performs a 'simulated' re-synthesis process by simply renaming an x86 file that was previously synthesized elsewhere. This way, the program behaves as if a re-synthesis was completed even without the ability to actually do so. The filenames to be used for the rename are handled in rebuild.c, which is basically just a skeleton program that could later be used to execute the synthesis script. Note that if re-synthesis is to be handled for more than one node, each node will need to have its own separate synthesis directory to preserve the status of previous synthesis processes.

The host demo software includes the ability to run multiple client nodes as well as send multiple publications at once to each client node. The idea is that sending more than one pub at a time reduces transfer overhead times as was seen with DMA transfers to the FPGA (this may not necessarily be the case though). When multiple pubs are sent to a client, they are all processed on the FPGA in a batch using a single DMA transfer. The results are then returned in a single transfer from each client and printed out. The only limitation on how many publications can be sent at once is the number of characters that the FPGA input FIFO will hold.

When the system first starts, it attempts to return to its previous state by restoring the software predicate database. This is done by re-adding any predicates that were only in software at

shutdown, and by re-deleting any predicates that were not yet removed from the hardware. At both shutdown and startup, the preds.xpath file represents the current state of the hardware; software predicate status is stored elsewhere.

This addresses some of the details of how various portions of the program have been implemented, and could prove useful when making future modifications. The first step after receiving raw publication data is to package it into 64-bit words and add the correct EOT characters. The message can then be sent to the FPGA, which will return information about the leaf values in each publication and the index values of hits. When more than one publication is sent to the FPGA at once, the returned results from each pub are concatenated together and returned in one transaction. It is important to note that for multiple pubs, the offset values returned by the FPGA are relative to the start of each pub, not the start of the entire block. The publication pointer is advanced to point to the start of the next pub for each check. After the information is returned from the FPGA, the unsure hits are checked alongside any predicates that are only in the software database, and then predicate IDs that are hits get packaged into a message and sent back to the server. Results from multiple pubs are concatenated together into one message to be sent back, and are separated by dashes.

To add a predicate, it must be in pcf format to be added into the predicate database on the system. Since predicates are expected to be available in an XPATH file, they must first be converted to the proper format. They are first loaded from the XPATH file into a 2-dimensional character array, and are then converted into an array of pcf type data structures. At this point the new predicates can be attached to the end of the existing predicate database structure. The IDs for the new predicates will be extracted from the XPATH file and will be attached to the predicates in the system.

Since predicates cannot actually be removed from the hardware until a re-synthesis occurs, deleted predicates are simply removed from the software database and ignored if their ID is returned by the FPGA. The array that matches that index values returned by the FPGA to the actual IDs of the predicates can be flagged to indicate which predicates have been removed, and they will be left out when the next rebuild occurs. The IDs of deleted predicates are also stored for future re-deletion in the event that the system is shutdown before they can be incorporated into a hardware rebuild.

When a rebuild is started, the client node updates its preds.xpath file to indicate the current state of its predicate database. A new FPGA image is then constructed using this XPATH file. When the re-synthesis is complete, the node can reload the FPGA with the new image, and update its predicate database to represent this new state. Any changes that were made after the rebuild was initiated will be implemented in the next rebuild.

It has been observed that occasionally the FPGA will get into a cyclic stall pattern, from which it does not recover (i.e. it parses one pub, stalls, and repeats). There have also been occasions where stalls have led to segmentation faults. The underlying cause for these problems is not known at this time. The issue seems to only occur when sending only one publication per transmission.

Section 4: Software – Single Node

The software for the single-node system is less complicated than for the multi-node system. After the .x86 image is synthesized, it is loaded into the FPGA, and publications are read from an input file and parsed. The system provides the option of reading and parsing more than one publication at a time. The host software requires three input files to run:

1) pe.x86 - image to load into the FPGA
2) predicates.pcf - a pcf formatted listing of all the predicates in the FPGA image
3) pubs.xml - an XML document containing publications

Note that the file naming conventions that applied to the multiple-node system are not used here. The exact names ‘pe.x86’ and ‘predinfo.pcf’ should be used for the input files.

During execution, not all nodes on the HHPC will successfully compile the host software due to library issues, so it may be necessary to try several nodes before finding one that will successfully build the executable. The FPGA does not always work on all nodes either, so if error messages are generated when trying to communicate with the FPGA, try a different node.

./test [XML_pub_file] [predicate_pcf_file] [num_pubs] [num_pubs_to_send]

The program will parse up to ‘num_pubs’ publications from the ‘XML_pub_file’ by sending them ‘num_pubs_to_send’ at a time to the FPGA.

For multiple publications, if the ‘num_pubs_to_send’ argument is greater than one, then the specified number of publications will be read from the input XML file, packaged, and sent to the FPGA in one DMA transaction. During packaging, each publication is separated by an EOT character with a string of three EOTs at the end of the group. All of the results will be concatenated together by the FPGA, and the host software will loop through until the entire response has been processed. It is important to not overfill the input FIFO of the FPGA by trying to send too many publications at once.